National Library of Canada
Canadian Theses Service
Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Services des thèses canadiennes

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL 339 (r. 86/01)

Canada

114

National Library of Canada | Bibliothèque nationale du Canada

Canadian Theses Division | Division des thèses canadiennes

Ottawa, Canada
K1A 0N4

0-315-23311-7

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

• Please print or type — Écrire en lettres moulées ou dactylographier

Full Name of Author — Nom complet de l'auteur

PRABHAT NARAYAN SAHAY

Date of Birth — Date de naissance

Oct 16th, 1958

Country of Birth — Lieu de naissance

INDIA

Permanent Address — Résidence fixe

Man-Mandir
Kamoo-2nd Road
Mithapur,
PATNA-800001, INDIA

Title of Thesis — Titre de la these

Heuristics for Selecting Best Paths for Testing Computer Programs

University — Université

University of Alberta

Degree for which thesis was presented — Grade pour lequel cette these fut présentée

M. Sc.

Year this degree conferred — Année d obtention de ce grade

1985

Name of Supervisor — Nom du directeur de these

Dr. Lee J. White

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHEQUE NATIONALE DU CANADA de microfilmer cette these et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la these ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

Oct 3rd, 1985

Date

Prabhat N Sahay

Signature

NL-91 (4/77)

The University of Alberta

Heuristics for Selecting Best Paths for Testing Computer Programs

by

© Prabhat Narayan Sahay

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Fall, 1985

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Prabhat Narayan Sahay

TITLE OF THESIS: Heuristics for Selecting Best Paths for Testing Computer Programs

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1985

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) Prabhat N Sahay

Permanent Address:

Man-Mandir
Kanoo-Lal Road
Mithapur
Patna 800001
India

Date October 1st 1985

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Heuristics for Selecting Best Paths for Testing Computer Programs** submitted by **Prabhat Narayan Sahay** in partial fulfillment of the requirements for the degree of **Master of Science.**

Dr. L. J. White

Supervisor

Dr. M. T. Ozsu

Dr. F. J. Pelletier

Dr. D. A. Szafron

Dr. J. T. Mowchenko

External

Date October 1st, 1985

To my parents

ABSTRACT

Many of the testing methods are forms of path analysis strategies, which require the selection of a set of paths along which testing will be conducted. Ideally, one would like to construct a set of tests which will detect all errors in a program. In general, the problem of finding such a set of test paths is known to be unsolvable. However, Zeil has developed a vector space measure which indicates those paths which best detect errors in a selected program predicate. This measure has to be applied to a set of paths post hoc, and so the problem is to heuristically characterize those paths which will provide maximal test information about the selected predicate; the problem of selecting the optimal set of paths is NP-hard. Experiments are conducted which utilize the vector space criterion to indicate those characteristics of paths which can best be used to test a given predicate. These characteristics will then provide a selection mechanism for an appropriate set of paths. A larger question involves the selection of those paths which sufficiently test all the program predicates, and will also be addressed by these experiments. A further issue is to experimentally characterize those program properties which lead to an irreducible error space for a predicate, and cannot be eliminated by any path through that predicate.

Acknowledgements

With sincere gratitude, I wish to thank my supervisor Dr. Lee J. White for sharing with me his vast knowledge in Program Testing. His guidance and encouragement have been invaluable.

I am also grateful to the members of my examining committee, Dr. M. T. Ozsu, Dr. F. J. Pelletier, Dr. D. A. Szafron and Dr. J. T. Mowchenko, for their helpful comments.

I wish to thank the National Science and Engineering Research Council of Canada for their support in the completion of this research.

Finally, I am indebted to the members of my family, especially my brother Pratap, for their support and encouragement.

Table of Contents

Chapter	Page
Chapter 1: Introduction	1
1.1. Why Testing?	1
1.2. Proposed Methodology	2
1.3. Thesis Organization	6
Chapter 2: A Model for Sufficient Path Testing and Perturbation Testing	7
2.1. Background	7
2.2. Basic Concepts	8
2.3. Basic Assumptions	13
2.4. Zeil's Model	19
Chapter 3: The Sufficient Path Testing System	24
3.1. Program Parse and Compile	26
3.2. Path Selection and Symbolic Execution	26
3.3. Sufficient Path Testing	28
Chapter 4: Experiments with Sufficient Path Testing on Linearly Domained Programs	32
Chapter 5: Analytical and Experimental Results	35
5.1. Unused Variables	36
5.2. Equality Predicates:	39
5.3. Invariant Expressions	41
5.4. Program Loops	43
5.5. Experimental Results	52
Chapter 6: Conclusions	62
6.1. Overview	62
6.2. Summary	62
6.3. Further Research	64
References	66
Appendix 1: Current Implementation of SPTEST	69
Appendix 2: Using the Sufficient Path Testing System	71
Appendix 3: Input Programs	73

List of Tables

Table		Page
4.1	Programs Upon Which SPTEST Experiments Were Conducted	34
5.1	Summary of Sufficient Set of Paths for Testing Predicate T_3	42
5.2	Summary of Sufficient Set of Test Paths for the GCF Program	49
5.3	Summary of Sufficient Set of Test Paths for the GCD Program	53
5.4	Results of Experiments Using SPTEST	54

List of Figures

Figure		Page
1.1	Domain Error	4
2.1	An Example of Program Partitioned into (C_j, T_j) Pairs	10
2.2	An Example of Directed Graph Representation of a Program	12
2.3	Examples of Assignment and Equality Blindness, Self-Blindness	18
2.4	An Example for Zeil's Results	21
3.1	An Overview of the SPTEST System	25
3.2	An Example of a Path Matrix for a Selected Path	30
5.1	An Example Program Flowchart	38
5.2	A Example Program (Program #3 in Table 4.1)	40
5.3	An Example Program Flowchart with 2^{12} Paths	44
5.4	Euclid's Algorithm for GCF	47
5.5	Euclid's Algorithm for GCD	51
5.6	Example of an Invariant Expression for Pragram #6	57
5.7	Example of a Loop Invariant for Program #2	58

Chapter 1

Introduction

1.1. Why Testing?

The goal of program testing is to gain confidence in software. Ideally, one would like to construct a set of tests which will detect all errors in a program. In general, the problem of finding such a set of tests is known to be unsolvable. Manna and Waldinger [12] have clearly summarized the theoretical barriers to complete testing:

"We can never be sure that the specifications are correct."

"No verification system can verify every correct program."

"We can never be certain that a verification system is correct."

Not only are all known approaches to absolute demonstration of correctness impractical, but they are impossible as well. Therefore, our objective must shift from an absolute proof to a suitably convincing demonstration of program correctness. The word **suitable,** if it is to have the same meaning to everyone, implies a quantitative measure, which in turn implies a statistical measure of software reliability. Our goal, then, should be to provide sufficient testing schemes to assure that the probability of failure due to hibernating bugs is sufficiently low to be acceptable. What is sufficient for a videogame is not sufficient for a nuclear reactor application. We can expect that each application area will eventually evolve its own software reliability standards. Concurrently, testing techniques will evolve to the point where it will be possible, on the basis of test results, to provide a quantitative prediction of the routine's reliability. But this is still in the future. For

now, it seems that all testing, in most applications, must be substantially increased before such measures can be applied and interpreted with confidence. In the past decade there is a growing agreement on the role of testing as a software quality assurance discipline.

The purpose of this research is to develop heuristics for selecting test paths for path-oriented testing strategies.

1.2. Proposed Methodology

In recent years a number of methods for automating portions of the software testing effort have been proposed. Many of these methods are forms of path analysis strategies [2,7,14,18], where the process of testing is treated as two operations:

1) selection of a path or set of paths along which testing is to be conducted;

2) selection of input data to serve as test cases which will cause the chosen paths to be executed.

Currently, the second stage of path analysis testing appears to be better understood. For general programs, the problem of generation of reliable test data is known to be **undecidable.** This means that it can be proven that no algorithm can be found in the general case for its solution. The test data selection problem involves the selection of a finite set of test data which reliably determines the correctness of a program over its entire input space. Howden [8] has shown the problem of **constructing** such a set for an arbitrary computer program is undecidable. For certain classes of programs, however, the domain testing strategy research [16-18] has shown that it is possible to implement reliable methods of selecting test data for a given path to detect specified types of errors.

A set of test data for testing a program construct for given path will be considered **reliable** if, whenever that construct behaves incorrectly along that path, the test set guarantees that incorrect output will be observed by the tester. A testing strategy is considered reliable if it generates only reliable test sets.

The work which has been done on the second stage of path analysis testing permits us the luxury of assuming that a reliable method of testing a selected path is available to us. This research is therefore concerned with the first stage of path analysis testing, the selection of test path.

Computer programs contain two types of errors which have been identified as **computation errors** and **domain errors** by Howden [8]. A path contains a **computation error** when a specific input follows the correct path, but an error in some assignment statement causes the wrong function to be computed for one or more of the output variables.

A **domain error** occurs when a specific input follows the wrong path due to an error in the control flow of the program. Figure 1.1 explains this for a program with exactly two variables. In this case the set of possible inputs to the program forms a flat plane. The bold line represents the correct border, and the dashed line indicates the border actually generated by the (incorrect) program. Input data points which fall within the regions between the two lines are associated with the wrong domain, causing the wrong path to be executed, creating a domain error.

Domain errors can occur due to an error in the program predicate, or due to an error in some assignment statement which subsequently affects the interpretation of a later predicate. The term **predicate error** has been employed to describe the former [7]. Similarly, errors in assignments will be termed **assignment errors** in this thesis. These are particularly interesting in that an assignment error might generate either a domain error or a computation error.

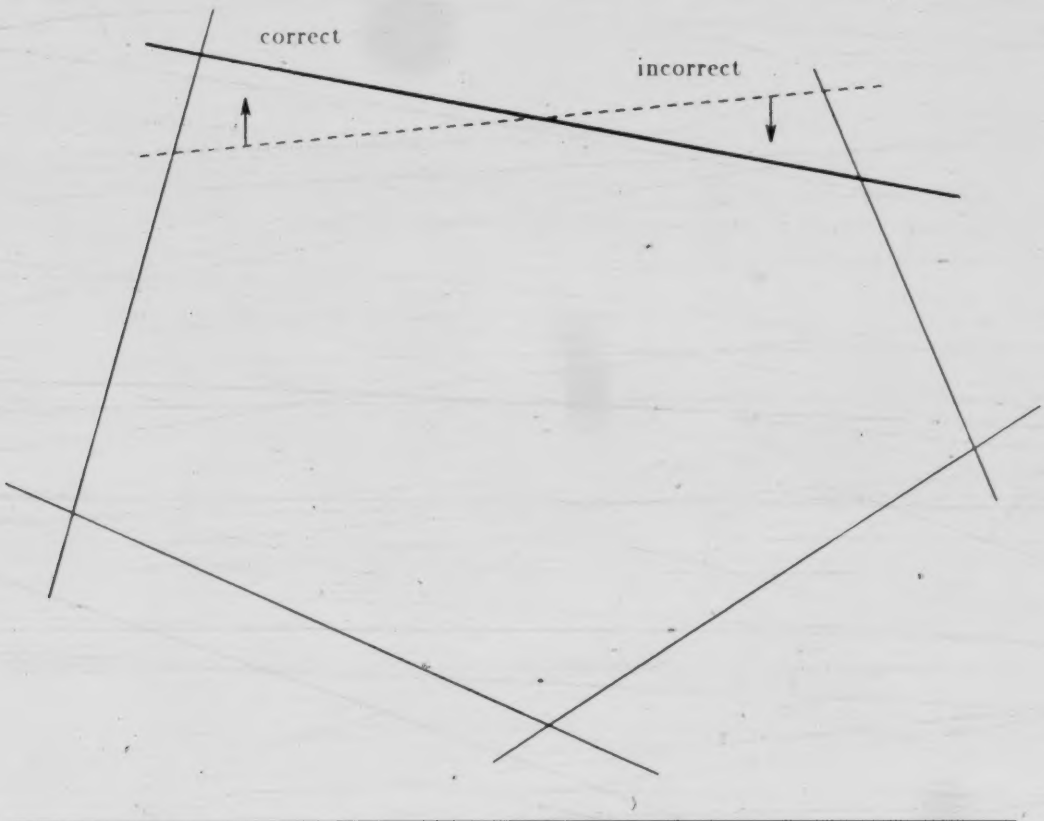

Figure 1.1 Domain Error

A final possibility is that a function and its subdomain may be missing altogether. This generally occurs when a programmer forgets that certain values are to be treated as separate or special cases. In the program this means that a corresponding predicate is missing and hence no constraint is generated to split some subdomain. These are called **missing path errors** [8].

This error classification is by no means exhaustive. Clearly, there exist other types of error such as syntax errors, errors in I/O, etc., which do not fall into these classes and do not bear a natural relational with the functional model of computer programs. Here we are attempting to capture the problems commonly thought of as "run-time" errors, faults which are not normally detectable by static checks such as compiler diagnostics nor by common hardware/software monitoring such as checks for division by zero or range checks.

Zeil has developed a vector space analysis technique called **perturbation testing,** and has applied it to both domain errors [22-24] and computation errors [25]. However, here his model is utilized for predicate errors (a subset of domain error), obtaining answers to the following kind of questions:

1) After a number of paths have been tested, what is the marginal advantage of choosing yet another path?

2) Is there a point at which we may say that no more paths need to be chosen through some program construct to be tested, i.e., that is has been **sufficiently tested** ?

A set of paths shall be considered to be a **sufficient set** for a program construct if the failure to detect some error in that construct, using a reliable method of selecting data points for those paths, implies that this error would go undetected for any path through the program.

1.3. Thesis Organization

Chapter Two summarizes Zeil's model for perturbation testing. In order to obtain a sufficient set of paths, a reliable test selection strategy must be utilized. If the errors are **domain errors,** then under certain conditions [3,18], the domain testing strategy can be shown to be reliable.

In Chapter Three, the details regarding the design and implementation of a computer system for selecting set of test paths for testing computer program predicates is elaborated. This system is based on the Zeil's model.

Chapter Four outlines the experimentation using the system described in Chapter Three.

Chapter Five presents the results from the experiments and discusses the issues involved in path selection.

Chapter Six draws conclusions from the experimental results. A discussion of possible further work follows the conclusions.

Chapter 2

A Model for Sufficient Path Testing and Perturbation Testing

In this chapter we shall summarize a model for predicate testing. This model was developed by Zeil and White [16,22] for predicate testing. It describes a class of programs whose domain borders are portions of linear hyperplanes. Zeil [24,25] has extended his model for perturbation testing.

2.1. Background

There are two major classes of variables used in a program. If a variable appears in a READ statement, it is classified as an **input variable;** all the other variables are called **program variables.** If a variable appears in a WRITE statement, it is classified as an **output variable.** An output variable can be either an input or a program variable.

The major components of most programs are computations and predicates. Computations assign new values of program variables, some of which may be sent directly as output while others are employed as intermediate results. Predicates are the decision functions of a program, commonly seen in "IF" statements or as the control statement of a loop.

Predicate functions often make use of program variables whose values have been determined by a previous computation along a particular path. The process of substituting for these program variables in terms of input variables along that path is called **interpreting** the predicate function, resulting in a **predicate interpretation.**

An interpretation of a simple predicate is **linear** in the input variables if and only if it is of the form

$$c_1v_1 + c_2v_2 + \ldots + c_mv_m \; \Phi \; k$$

where $v_1, \ldots, v_m$ are the input variables, $c_1, \ldots, c_m$ and k are constants. Φ represents one of the relational operators ($<$, $>$, $=$, $\leq$, $\geq$, $\neq$).

2.2. Basic Concepts

The central element in this model is the **environment.** The environment of a program represents the values of all inputs and variables at any point in the program's execution. The environment may be represented as the following vector:

$$\underline{x} = (x_1, \ldots, x_m, y_1, \ldots, y_n)^T$$

The y_i represent the program variables, whose values are established by the computations along a chosen path through the program. The x_i represent the values of input variables. For reasons of primarily notational convenience, the number of input variables is treated as a constant. Initially, only the input variables are considered to be defined. The program must define the program variables as functions of the input values and constants.

The components of the program itself can then be described in terms of their interactions with the environment vector. A program is considered as a set of pairs of the form (C_j, T_j) where C_j is a computation or transformation to be applied to the current environment to generate the new environment and T_j is a predicate function which is applied to the new environment and then compared with zero to determine the index of the next (C_j, T_j) pair. These predicates may be either equalities or inequalities. In effect, this means that a program is composed of

statement pairs of the form

$$\underline{x} = C_j(\underline{x})$$

and

if $T_j(\underline{x})\ \Phi\ 0$ then go to j'

else go to j''

where Φ is any relation operator. Note that a single C_j can correspond to an entire block of assignment statements.

We shall let (C_0, T_0) designate the start of the program. An example of partitioning a program into (C_j, T_j) pairs for an Integer Division Remainder program is given in Figure 2.1.

For example, in a program with three input variables X, Y and Z, and two program variables A and B, the environment vector would contain five entries corresponding to

$$\underline{x} = (X, Y, Z, A, B)^T$$

The computation corresponding to the program fragment,

```
A = B - Z + 2*X
B = Y + 5
IF A > B THEN .....
```

is the function

$$C(\underline{x}) = \begin{vmatrix} v_1 \\ v_2 \\ v_3 \\ v_5 - v_3 + 2v_1 \\ v_2 + 5 \end{vmatrix}$$

```
READ, X, Y
R = 0
A = 0                                  (C0, T0)
IF (X .GE. 0) THEN DO
  IF (Y .GT. 0) THEN DO                (C1, T1)
    R = X                              (C2, T2)
    WHILE (R .GE. Y) DO                (C3, T3)
      A = Y                            (C4, T4)
      WHILE (R .GE. A) DO              (C5, T5)
        R = R - A
        A = A + A                      (C6, T6)
      END WHILE
    END WHILE                          (C7, T7)
  END IF
END IF                                 (C8, T8)
PRINT, 'Remainder is ', R
STOP
END
```

Figure 2.1 An Example of Program Partitioned into (C_j, T_j) Pairs

The predicate function for the IF statement ending that fragment would be $T(\underline{x}) = v_4 - v_5$, and the result of this expression would be compared to zero to choose the subsequent path.

A program can be represented as a directed graph G = (V, A), where V is a set of nodes and A is the set of arcs or directed edges between nodes. The directed graph representation of a program will contain a node for each occurrence of a (C_j, T_j) pair, and an arc for each possible flow of control between these pairs. An example of the directed graph representation for the Integer Division Remainder program is given in Figure 2.2.

A **walk** in a digraph is defined as an alternating sequence of nodes and arcs $(v_1, a_{12}, v_2, a_{23}, \ldots, a_{k-1,k}, v_k)$ such that each arc $a_{i,j}$ is from node v_i to node v_j. A **path** is then defined to be a walk in the directed graph which begins at (C_0, T_0) and ends at a valid HALT statement. Two walks which differ only in the number of times of a particular loop in the program is executed are differentiated as two distinct paths. Thus the number of paths in a program can be infinite [17]. The term **subpath** will be used to designate a path which begins at (C_0, T_0) and does not end at a valid HALT statement.

The process of executing a program will therefore consist of choosing a sequence of (C_j, T_j) pairs constituting a legal path through the program, and applying the successive transformation to the environment. For example,

$$\underline{x}_{k+1} = C_k \mathbf{o} C_{k-1} \mathbf{o} \cdots \mathbf{o} C_1 \mathbf{o} C_0(\underline{x}_0)$$

where $\underline{x}_0$ is the initial environment and **o** denotes the composition of functions. Frequently we shall group these computations together and define C_A as the total computation along a path P_A:

$$C_A = C_k \mathbf{o} C_{k-1} \mathbf{o} \cdots \mathbf{o} C_1 \mathbf{o} C_0.$$

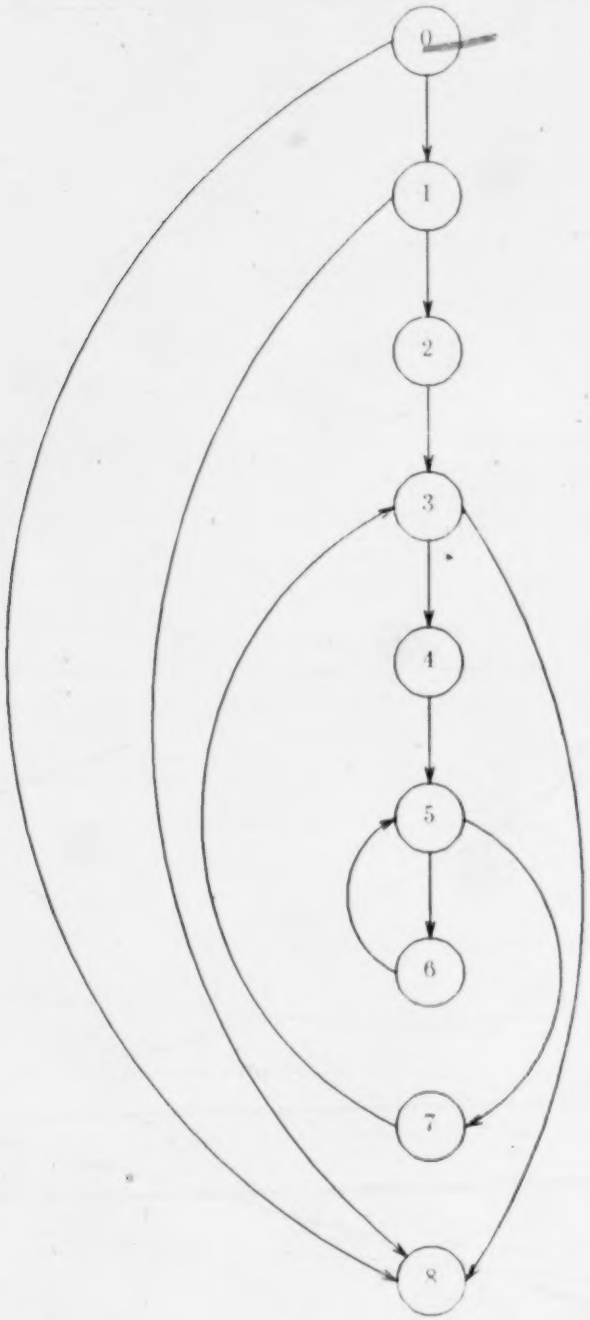

Figure 2.2 An Example of Directed Graph Representation of a Program

Then the environment after executing the program along some path P_A would be given by $\Sigma_A = C_A(\Sigma_0)$

As execution of a program proceeds along some path, every predicate encountered places restrictions on the subdomain of input points which will cause that path to be executed. However, the constraints imposed by equality predicates are fundamentally different from those imposed by inequalities, since a valid predicate reduces the dimension of the space of legal input points. Equality restrictions can arise from sources other than equality predicates. Combinations of two or more inequalities may coincide to impose an equality restriction on the input domain. An example of this would be the pair "IF $A \leq B$" and "IF $A \geq B$". If both of these predicates are taken as true, then the condition "$A = B$" is implied. Equality conditions arising from such combinations will be referred to as **coincidental equalities.** Since a path can actually involve many equality restrictions, it will be necessary to collect a set of equality restriction vectors, $\{r_i\}$, for each path.

The path in question is **feasible** if there exists some input data which causes that path to be executed.

2.3. Basic Assumptions

A set of limiting assumptions about both paths and programs are necessary in order to formulate reliable testing strategies. We will be concerned with **feasible** (sub) paths leading from the beginning of the program up to and including the program construct being tested. A subpath P shall be called **testable subpath** if there exists a path P′ such that

1. P′ ends with a valid HALT statement;

2. There exists some input value causing the path $P''' = (P, P')$ to be executed.

In this model the programs are represented in terms of computation – predicate pairs (C_j, T_j) with execution along a path being described in terms of the environment $\underline{x}$ and the equality restrictions $\{r_j\}$. The conditions on programs which we shall analyze in this research are as follows:

(1) missing path errors do not occur;

(2) the input space is continuous;

(3) predicates are simple, not combined with AND, OR, or other logical operators;

(4) adjacent domains compute different functions.

The first assumption is required because no testing strategy based only on the program text can guarantee detection of missing path errors. The assumption of continuity permits the use of standard mathematical tools. In practice this should cause little difficulty as long as the size of the domain is not comparable to the discrete resolution of the space [16]. The third assumption is convenient, simplifying the functional forms of the predicates, but need not actually restrict the set of acceptable programs, since any program can be easily transformed to eliminate compound predicates. The final assumption simply states that, if a domain error occurs, there must exist some point in at least one of the affected domains which produces incorrect output.

Even given these assumptions, the problem of determining the correctness of program predicates remains unsolvable. Some knowledge of the class of permissible functional forms for the predicates is required. Therefore the next assumption is:

(5) predicate interpretations are linear in the input variables.

Any program satisfying these five constraints will be called a **linearly domained program,** and it will be these type of programs which will be studied in this thesis. One characteristic of linearly domained programs is that the **feasibility** of paths can be determined.

Zeil has generalized the class of programs for which his model can be applied; consider the following three constraints:

(6) all program computations fall within some class $\{C_i\}$ which is closed under functional composition.

(7) all predicates fall within some class $\{T_i\}$ which is closed under composition with $\{C_i\}$.

(8) $\{T_i\}$ is a vector space of dimension k over R.

The class $\{C_i\}$ represents the set of possible computation functions for use in the program. The set of possible predicate interpretations is represented by the class of function $\{T_i\}$.

Programs satisfying the conditions (1) through (4) and (6) through (8) will be called **vector bounded programs.** Examples of useful vector spaces of functions include the set of all linear functions, the set of polynomial functions of degree k, and the set of multinomials of degree k. Thus linearly domained programs are a special case of vector bounded programs.

The pivotal concept of this model is the definition of the set of predicate interpretations as a finite-dimensioned vector space. A vector space provides a powerful way of describing and manipulating an infinitely large set of functions.

Two properties of vector spaces are particularly significant here. First, any finite-dimensional vector space can be described by specifying a finite set of characteristic vectors, such that any member of the vector space can be formed from a linear combination of these characteristic vectors. Second, a vector space is closed under the operations of addition and scalar multiplication. Suppose that some correct predicate T has been replaced by an erroneous form, T'. Then the expression "T − T'" represents the error term in the incorrect predicate. Since the vector space containing T and T' is closed under subtraction, the error term "T − T'" must also be in the same vector space.

Define the function e as an error term for which

$$T' = T + \alpha e \qquad \alpha \neq 0$$

Then the interpretation of the incorrect predicate T' along an subpath ending at T' with final environment $\underline{x}$ is given by

$$T'(\underline{x}) = T(\underline{x}) + \alpha e(\underline{x})$$

If a reliable method of testing a given path is employed, then a predicate error is detectable whenever the interpretation of the incorrect predicate is not a multiple of the correct predicate's interpretation. Thus a predicate error is undetectable along a given path if and only if there exists a positive scalar h such that

$$T(\underline{x}) = hT'(\underline{x})$$

for all $\underline{x}$ in the path domain. Define **Null(C)** as the set

$$\{e: e \in \{T_i\} \text{ and } \forall\, \underline{x},\ e \circ C(\underline{x}) = 0\},$$

the set of all error terms in $\{T_i\}$ which are identically zero on the path where C is computed. Null(C) consists of the set of functions which evaluate to zero when the expressions computed in C are substituted for the program variables. For example,

after the assignment statement $X = f(\underline{x})$, the expression $X - f(\underline{x})$ can be added to a predicate without detection. Of course, if a different test path is selected, X may be assigned a different expression and the error term $X - f(\underline{x})$, being nonzero, could be detected. This behavior, associated with the values currently assigned to each program variable, is called **assignment blindness.** An example is shown in Figure 2.3, where the two versions of test predicate are indistinguishable as long as the assignment A = 1 is unchanged.

A similar behavior is associated with equality restrictions on a selected path. An example is shown in Figure 2.3, where the two versions of second predicate are indistinguishable as long as the first predicate is true. Hence, the set of equality restrictions $\{r_i\}$ is included in the set of undetected errors to acknowledge **equality blindness.**

The final component of the set of undetectable predicate errors is the predicate T′, and is termed **self-blindness.** It is an interesting property of predicate testing that a predicate can never be distinguished from its multiples, since $T(\underline{x})$ compared with zero and $\alpha T(\underline{x})$ compared with zero are identical for all positive real numbers α. An example of this is given is Figure 2.3.

For convenience, the set of expressions denoting the assignment, equality, and self-blindness spaces for any path P_A will be denoted by $BLIND(P_A)$.

The vectors $v_1, v_2,\ldots,v_r$ in a vector space V are said to **spanV** if every vector in V is expressible as a linear combination of $v_1, v_2,\ldots,v_r$. That is, for every $v \in V$ there are scalars $a_1, a_2,\ldots,a_r$ such that

$$v = a_1v_1 + a_2v_2 + \ldots + a_rv_r$$

Assignment Blindness

Correct	Incorrect
A = 1	A = 1
.	.
.	.
IF B > 0 THEN	IF B + A > 1 THEN
.	.
.	.
.	.

Equality Blindness

Correct	Incorrect
IF D = 2 THEN	IF D = 2 THEN
.	.
.	.
IF C + D > 3 THEN	IF C > 1 THEN
.	.
.	.
.	.

Self-Blindness

Correct	Incorrect
X = A	X = A
.	.
.	.
IF X - 1 > 0	IF X + A - 2 > 0
.	.
.	.
.	.

Figure 2.3 Examples of Assignment and Equality Blindness, Self-Blindness

2.4. Zeil's Model

In reference [23], Zeil then demonstrates the following four results:

(a) Let P_A be a testable subpath in a vector bounded program. Let C' be the function computed along that path, T' be the final predicate in P_A, and $\{r_i\}$ be the set of equality restrictions on the domain of P_A. Then an error e in T' will be undetectable on P_A if and only if

$$e \in \text{span}[\, \text{Null}(C),\ \{r_i\},\ T' \,]. \tag{1}$$

(b) If a set of testable subpaths $\{P_i\}$, all ending at some predicate T', has been reliably tested, then a subpath P_A also ending at T' need not be tested if an only if

$$[\, \bigcap_i \text{BLIND}(P_i) \,] \subseteq \text{BLIND}(P_A) \tag{2}$$

(c) A set of testable subpaths $\{P_i\}$, all ending at some predicate T' is sufficient for testing T' if

$$[\, \bigcap_i \text{BLIND}(P_i) \,] = \{T'\} \tag{3}$$

(d) A minimal set of subpaths sufficient for testing a given predicate in a vector bounded program will contain at most k subpaths, where k is the dimension of $\{T_i\}$.

This last result involving the minimal set of sufficient subpaths can be considerably simplified for the case of linearly domained programs:

A minimal set of subpaths sufficient for testing a given predicate in a linearly domained program will contain at most $m+n+1$ subpaths, where m is the number

of input variables and n the number of program variables.

To demonstrate these results, consider the program shown in Figure 2.4. This program computes the greatest common divisor of any two positive integers. The predicates are labeled in order to provide reference points for describing paths. Paths will be specified by listing the predicate names in the order in which they are encountered, followed by **t** or **f** to indicate the branch chosen according to the truth or falsehood of the predicate. Subpaths will be denoted by listing all predicates up to and including the ending predicate, with the ending predicate marked with **?** since no branch has yet been chosen following that predicate.

The environment vector for this program will have five elements, corresponding to $(I, X, Y, A, B)^T$. Let's start the test path selection process for predicate T_1 by examining the shortest possible test path (T_1:?) leading up to T_1. The subpath P_1 (T_1:?) consists of the first four program statements. The symbolic execution along the subpath P_1 is

$A = X$

$B = Y$

The assignment blindness vectors are therefore given by the expressions "X − A" and "Y − B". The self blindness vector for this subpath is based on the expression "B − A" and its interpretation in terms of input variables is "Y − X". There are no equality restrictions. Therefore $BLIND(P_1)$ consists of the expressions "X − A", "Y − B" and "Y − X". Hence if testing along the subpath (T_1:?) does not reveal an error, the predicate T_1, whose functional part here is "B − A", may be incorrect with the correct form being "X − A", "Y − B", "Y − X", or any of the infinite expressions which can be obtained from combinations of the three characteristic errors which cannot be detected using this test path.

```
          PROGRAM GCD
          READ, X, Y
          A = X
          B = Y
(T1)      WHILE (A .NE. B) DO
(T2)        WHILE (A .GT. B) DO
              A = A - B
            END WHILE
(T3)        WHILE (B .GT. A) DO
              B = B - A
            END WHILE
          END WHILE
          PRINT, X, Y, A, B
          END GCD
```

Euclid's Algorithm for GCD

For this program the following set of paths are sufficient to test predicate T_1:

1. $(T_1{:}?)$
2. $(T_1{:}t, T_2{:}t, T_2{:}f, T_3{:}f, T_1{:}?)$
3. $(T_1{:}t, T_2{:}f, T_3{:}t, T_3{:}f, T_1{:}?)$

Figure 2.4 An Example for Zeil's Results

Next consider the subpath P_2 (T_1:t, T_2:t, T_2:f, T_3:f, T_1:?). The symbolic execution along this subpath gives

$A = X - Y$

$B = Y$

With no equality restrictions, the predicate interpretation for the ending predicate is "2Y − X". Therefore BLIND(P_2) consists of the expressions "X − Y − A", "Y − B" and "2Y − X". The set of undetectable error using these two paths is given by the intersection of BLIND(P_1) and BLIND(P_2), which is given by the expressions "A − Y" and "B − Y".

Next consider the subpath P_3 (T_1:t, T_2:f, T_3:t, T_3:f, T_1:?). The symbolic execution along this subpath gives

$A = X$

$B = -X + Y$

There are no equality restrictions and the predicate interpretation for the ending predicate is "Y − 2X". Therefore BLIND(P_3) is given by the expressions "X − A", "Y − X − B" and "Y − 2X". The intersection of BLIND(P_3) with the total undetectable space after the first two paths is given by the expression "B − A". Therefore the total undetected space has been reduced to the self blindness vector and from Zeil's third result we know that this error term can never be eliminated from the undetected space. So paths P_1, P_2, P_3 are sufficient to test predicate T_1.

Testing predicate T_2 is now quite simple. There are no assignment statements between T_1 and T_2, so the assignment blindness for both predicates are the same. Both predicates have the same functional form, "B − A", so the self blindness vector is unchanged. As with T_1, there are no equalities to be concerned with. The net result is that extending each of the test paths for T_1 by assuming T1 is true

gives a path with the same blindness space for T_2 as for T_1, so the same set of paths form a sufficient test set for T_2.

Similarly by extending each of the test paths for T_1 by assuming T_1 is true and T_2 is false gives a path with the same blindness space for T_3 as for T_1, so the same set of paths is also sufficient for testing T_3.

In Chapter Four, experiments will be conducted entirely on linearly domained programs. Associated with each predicate to be tested will be an **error space** of maximum dimension (m+n+1). As each subsequent subpath through that predicate is chosen for testing, the resultant error space is reduced in dimension (or else that subpath is discarded and not tested). It is specifically the purpose of this research to develop heuristics for the choice of paths to effect rapid reduction of predicate error spaces. In order that these paths be viable for path testing, it is understood throughout that the selected paths must be feasible. In the next chapter a system for selecting a set of test paths for testing program predicates will be discussed.

Chapter 3

The Sufficient Path Testing System

The computer system which obtains sufficient test paths for selected predicates is called SPTEST. The purpose of SPTEST is to evaluate a proposed path for testing a given predicate by obtaining the reduction in the predicate error space due to that path. SPTEST does not depend upon the domain testing strategy [16-18] nor does it assume that domain testing will be used to test program predicates. It simply selects a set of test paths appropriate for testing computer programs. Once a set of paths is selected then any reliable testing strategy can be used for preforming the actual testing along the selected paths.

The overall structure of SPTEST is given in Figure 3.1. The input for this system is an ANSI FORTRAN program. The input program is assumed to be syntax error free and **linearly domained;** any number of predicates can be non-linear, but these cannot be involved in a path to be analyzed. This system is written in FORTRAN 77 and presently runs on UNIX and MTS (Michigan Timesharing System, similar to IBM 370 VM). The system is about about 6000 lines.

In order to evaluate the usefulness of a path, SPTEST requires some information about the path just selected. This includes the total number of program and input variables, the test predicate interpretation, program variable interpretations and any equality restrictions encountered along the path. This information is provided using the first two phases of DOMAIN TESTING SYSTEM – III (DTS-III) [21]. The first phase of DTS - III (Program Parse and Compile) was not altered for this purpose, but in the second phase (Path Selection and Symbolic Execution) a slight modification was needed in order to interface

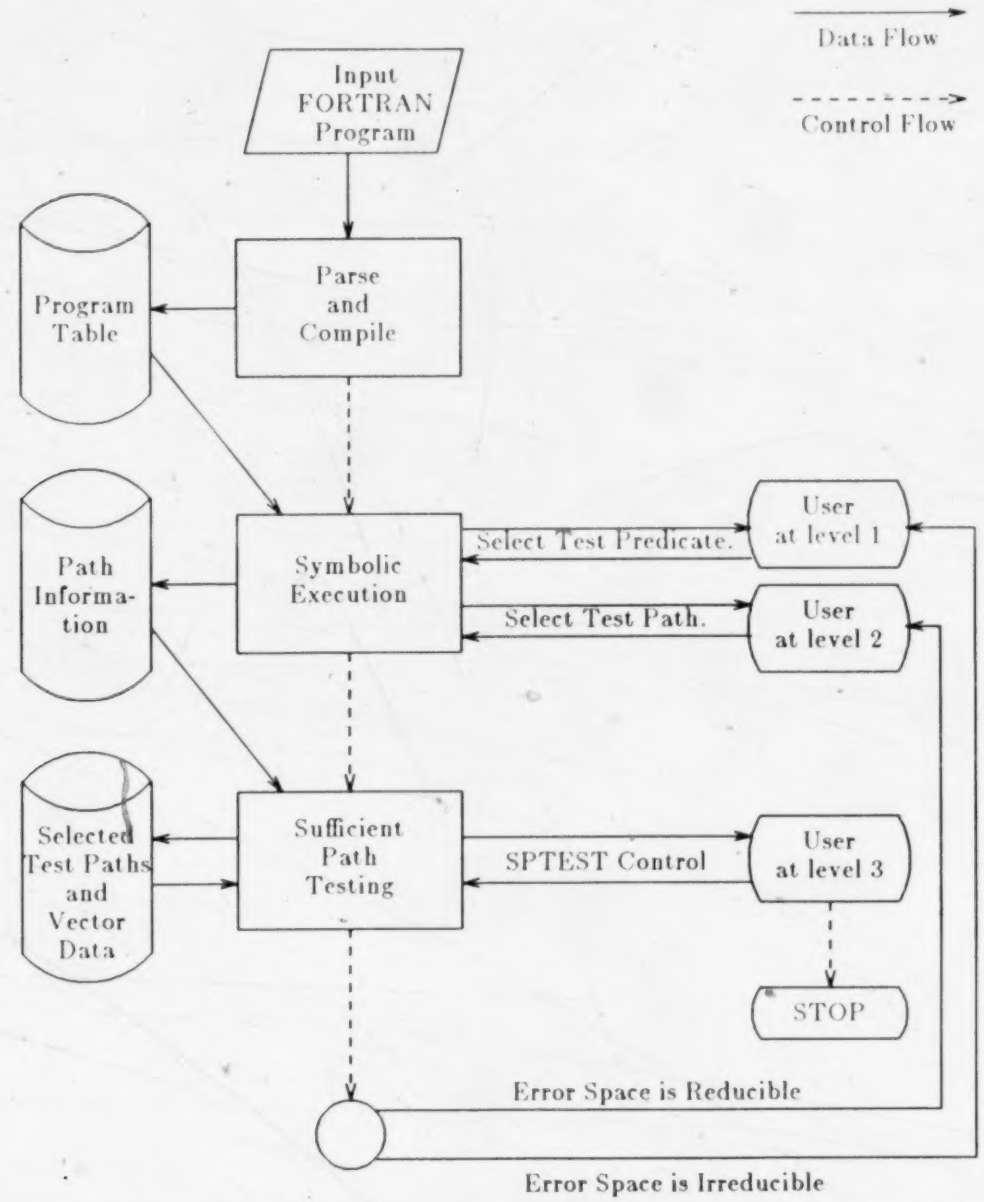

Figure 3.1 An Overview of the SPTEST System

with SPTEST. The whole system consists of three phases:

1) Program Parse and Compile

2) Path Selection and Symbolic Execution

3) Sufficient Path Testing

3.1. Program Parse and Compile

The first phase of this system is a parse of the submitted program. One aspect of the parse relates to input variables and program variables; from the declaration data and program syntax, a symbol table is constructed where these two categories of variables are differentiated. Critical aspects of the control flow are obtained by parsing for true or false predicates (from IF THEN or GOTO constructs), for labels, and for iteration loops (from DO LOOP or WHILE DO constructs). Predicate lists and other control information is secured from these parses. Arithmetic expressions for either assignment statements or predicate expressions are stored as binary trees. All of this data from the parse is referred to as a program table file in Figure 3.1.

The compilation program determines the linearity of all assignment and predicate expressions. Non-linear expressions and program variables are flagged for future reference, whereas linear expression and program variables are put in a standard form for later processing.

3.2. Path Selection and Symbolic Execution

This is the second phase of the system. At the beginning of this phase the user is asked which predicate is to be tested. The user indicates the desired predicate. Then the user is asked to select a path for testing that predicate. This is specified

interactively by indicating a true or false decision whenever a predicate is encountered along the path, or in the case of iteration loops, the number of times an iteration loop should be executed is specified. The collection of all these decisions then specifies the path for the system.

The symbolic execution of this path is carried out by evaluating the sequence of assignment statements and predicates occurring in the path, and substituting for all program variables and assignments in terms of symbolic input variables as the path is traversed. This is in effect execution without data, where read statements merely define the variable to be read as an input variable, and all program variables and predicate expressions are given symbolically in terms of input variables and constants.

An example of symbolic execution follows:

```
      READ(5,100) VAR1, VAR2, VAR3
100   FORMAT(3F10.2)
      SUM1 = VAR1 + VAR2
      SUM2 = VAR2 + VAR3
      SUM3 = VAR1 + VAR3
      RESULT = 2.0 * SUM1 + SUM2 - SUM3
      WRITE(6,110) RESULT
110   FORMAT('  Result = ', F10.2)
      STOP
      END
```

When the first line is executed, VAR1, VAR2, and VAR3 are defined as input variables and their symbolic values are defined as just their variable names. The second line is ignored. The third line causes SUM1 to be defined as 1.0*VAR1+1.0*VAR2. The fourth and fifth lines define SUM2 and SUM3 similarly. The sixth line will define RESULT as 1.0*VAR1+3.0*VAR2.

So what is needed in the system is a method of keeping track of values of program variables in terms of the input variables. There are three different ways of

doing this. The first is to keep a stack of all assignments as they occur without resolving them (a history so to speak) and then perform the operations when an interpretation is needed. The second approach is to reduce each new assignment using reverse Polish notation and store that symbolic expression in tree form. These two techniques are generally used when the assignment statements have non-linear expressions. A third technique, and the one used in this system, is to maintain a set of tables which will have either the coefficient of the input variables and a constant or a non-linearity flag for every active program variable. The non-linearity flag is set whenever a program variable is to be assigned an expression that can not be reduced to the form $c_1 \times x_1 + c_2 \times x_2 + \ldots + c_m \times x_m + c$. Therefore, any assignment involving this program variable is automatically flagged as non-linear.

At this point constraints are added which bound the input variables, related to limits their values may assume. These are parameters which are set once and for all given the storage and other software characteristics of the host installation. All these constraints are then submitted to a linear programming package, where redundant constraints are identified and eliminated. Now it can be determined whether the selected path is feasible or not. This decision can be made in general because we have assumed that we are working with linearly domained programs. If the path is not feasible, the user is so informed, and selects another path. If the selected path is feasible, the system proceeds to the next stage of sufficient path testing.

3.3. Sufficient Path Testing

Once a path is selected, then path information is passed to SPTEST. SPTEST also uses interactive input from the user to control the flow of the model. The user

is asked if this is the first path for the selected predicate to be tested; if so, then a path matrix is constructed of the transformations generated along this path. There is a row in the matrix for each program variable defined in the program which describes the assignment blindness for this path, for each equality restriction encountered along the path to include the equality blindness, and a row for the target predicate to include the self blindness for this path. In this matrix, there is a column for a constant, a column for each input, and a column for each program variable defined in the program. Figure 3.2 illustrates a path matrix for a specific program example. The assignment blindness vectors for subpath (T_1:t, T_2:t, T_3:t, T_3:?) corresponds to the first four rows of the path matrix. The equality restriction encountered along this subpath corresponds to the fifth row of the path matrix and the self−blindness corresponds to the last row. Therefore, the undetectable error space for this subpath is given by the rows of the this matrix. In order to evaluate this path we need to solve the equation (1) from Chapter 2; equation (1) requires the calculation of the restriction equations for the space spanned by the path matrix, after which this set of restriction equations are written out to an external file. The control of the system returns to the symbolic part of the system, and the user is asked to select another path.

If this is not the first path for the chosen predicate, then the stored information regarding previously selected test paths for this predicate is read from an external file. The path matrix for the new path is constructed as described above, and the restriction equations for the space spanned by the path matrix is calculated, and added to the independent restriction equations from all previously tested paths through the predicate as indicated by equation (2) from Chapter 2. In order to accomplish the intersection operation in equation (2), an independent set of

```
            READ, X, Y
            C = 0
            D = 0
            E = 0
            I = 0
(T1)        IF (X .GT. Y) THEN DO
              C = Y + 1
            ELSE DO
              C = Y - 1
            END IF
(T2)        IF (C .EQ. 0) THEN DO
              D = 2*X + Y
            END IF
            I = 1
(T3)        WHILE (I .LE. Y) DO
              E = E + 2*I
              I = I + 1
            END WHILE
            PRINT, E
            STOP
            END
```

In this program, after the execution of subpath (T_1:t, T_2:t, T_3:t, T_3:?) the program variables, equality restriction (r_1) and the final predicate (T_3) will have the following symbolic values: $C = Y + 1$, $D = 2X + Y$, $E = 2$, $I = 2$, $r_1 : -Y - 1$, and $T_3 : Y - 2$.

	Const	X	Y	C	D	E	F	I
C	1	0	1	-1	0	0	0	0
D	0	2	1	0	-1	0	0	0
E	2	0	0	0	0	-1	0	0
I	2	0	0	0	0	0	0	-1
r_1	-1	0	-1	0	0	0	0	0
T_3	-2	0	1	0	0	0	0	0

Figure 3.2 An Example of a Path Matrix for a Selected Path

restriction equations is obtained. If there is no increase over the number of restriction equations from previous paths, then the path under consideration does not effectively reduce the error space further, and should be discarded. If there is an increase in the number of independent restriction equations, then the error space is correspondingly reduced, and this new set of restriction equations is written out to the external file. If the resulting undetected error space is irreducible, then a message is written out to the user indicating the construction of a sufficient set of test paths. The user is asked either to select another predicate for testing or to terminate the testing process. Zeil [22,23] has argued that self-blindness can actually never be eliminated, so the minimum dimension of a final error space is at least one but in general can be larger (see equation (3) from Chapter 2). We can determine the **irreducible error space** of a predicate to be that which cannot be reduced further by any path through that predicate. Another objective of the experiments in Chapter 4 will be to characterize this irreducible error space, so as to provide a better stopping criterion for the procedure of selecting paths and evaluating the resulting error spaces.

In the next chapter experimentations will be outlined using this system.

Chapter 4

Experiments with Sufficient Path Testing on Linearly Domained Programs

Experiments using SPTEST with the supporting DTS-III system have been conducted with a number of linearly domained programs in order to obtain greater insight into path-oriented testing methods. Since SPTEST provides a vector space metric which evaluates the effectiveness of a set of paths for testing a predicate post hoc, these experiments could provide information as to the initial selection of the paths. For example, with DTS-III this information could be used to aid the user in the interactive selection of paths, or to provide an option of automatic selection of paths with this system. Another overall objective of the study is to better characterize the irreducible error space for each predicate. This would not only improve our conceptual understanding of the path testing process, but provide a substantially improved stopping criteria for using SPTEST to select a sufficient set of paths to test a given predicate or all the predicates in a program.

The specific questions we have attempted to resolve with this study are:

(1) Is it possible to find a minimum set of paths for testing a given predicate?

(2) The upper bound on the number of paths for testing a predicate is (m+n), where m and n are the number of input and program variables respectively. How many paths are required to test each predicate in practice?

(3) To what extent is it possible to reduce the number of paths for testing a predicate by carefully choosing the first path? If so, how can we decide which path to select first?

(4) The upper bound on the number of paths for testing all the predicates in a program is $p(m+n)$ paths, where p is the number of predicates. How many paths are required to test all of the predicates in a program in practice?

(5) To what extent is it possible to reduce the number of paths for testing all the predicates by carefully choosing the first predicate? If so, how can we decide which predicate to select first?

(6) In practice, how large is the irreducible error space for a predicate after a sufficient set of paths have been chosen? To what extent can we characterize both the vectors in this space and the corresponding program features so as to be able to a priori predict its dimensionality?

Table 4.1 shows the characteristics of the linearly domained programs upon which the SPTEST experiments were conducted. Note from the functionalities of these programs that they are diverse and non-trivial, illustrating that linearly domained programs are an important class and worth investigating as a first step. The complexity of these programs is also quite diverse, with five different and commonly accepted complexity measures given for each program.

MaCabe's cyclomatic complexity measure, V(G) is defined as follows:

$$V(G) = e - n + 2 \tag{4}$$

where n is the number of nodes and e is the number of edges in a directed graph G [13]. In Chapter 2 we have shown that a program can be represented as a directed graph. Therefore using a directed graph representation of programs given in Table 4.1 and equation (4), MaCabe's cyclomatic complexity measure is calculated.

Pgm.	Function	Lines	Error Space			# of Pred.	Iteration Loops		McCabe Complexity Measure
			Input m	Pgm. n	m+n+1		#	Nesting Level	
1	Euclid GCD	13	2	2	5	3	3	1	4
2	Integer Round-up	15	1	3	5	2	1	-	3
3		15	2	1	4	3	0	-	4
4	Integer Division Remainder	18	2	2	5	4	2	1	5
5	Euclid GCF	19	2	3	6	3	1	-	4
6	Conditional Series Summation	27	2	5	8	4	1	-	5
7	Sorted Set Intersection	31	8	5	14	6	2	1	6
8	Binary Search	42	11	5	17	7	1	-	9
9	Sorted Set Union	65	12	8	21	9	3	-	10

Table 4.1 Programs Upon Which SPTEST Experiments Were Conducted

Chapter 5

Analytical and Experimental Results

In this chapter analytical and experimental results obtained from the experiments done with SPTEST will be discussed. These results have provided a greater insight into the path selection process.

It will not be possible to obtain a minimum set of paths through a predicate in general. If iteration loops are involved, there are potentially an infinite number of paths to examine. Even without loops, this problem to still NP-hard, as the number of paths can grow exponentially, and a minimum set cannot in general be selected.

In the course of this research we learned that if one carefully selects the first path through a predicate, one can considerably improve the bound on the number of the paths required to sufficiently test that predicate. This can be stated formally by the following theorem:

Theorem 1:

The minimum sufficient set of subpaths required for testing a given predicate in a linearly domained program will contain at most (n+e+1) paths, where n is the number of program variables and e is the number of independent equality restrictions encountered along the first path chosen for that predicate.

Proof:

The initial dimension of the error space for the given predicate is (m+n+1) for a linearly domained program, where m is number of input variables. After the first path has been chosen, the remaining error space due to assignment blindness is of dimension at most n, the space due to equality blindness is of dimension at most e,

and the space due to self-blindness is of dimension one. Therefore the total untested error space after testing the first path will be at most dimension n+e+1. Here if e≥m then this path would be rejected under equation (2) from Chapter 2, since it does not reduce the initial dimension. In constructing a sufficient set of test paths, any subpath which fails to reduce the dimension of the error space by at least one would be rejected. Thus after testing two paths the dimension of the error space will be at most n+e. Continuing in this fashion it is clear that after choosing the first path the number of paths required to test a predicate is at most n+e+1 minus the dimension of the irreducible error space, and from equation (3) from Chapter 2, the irreducible error space for the predicate is of at least dimension one. Thus the total number of sufficient paths will be no more than (n+e+1).□

Both analytical work and experiments have established that there will be three sources of vectors in the irreducible error space other than the self-blindness of the predicate itself:

1) unused variables,

2) equality restrictions, and

3) invariant expressions.

5.1. Unused Variables

In SPTEST all the variables (both input and program variables) should be defined in the beginning of the program. It might be possible that in testing some predicate some of the variables will be defined but not used. In these cases one would like to know how this affects the final dimension of the error space.

Now consider the program flowchart in Figure 5.1.

In this program consider a variable X which is only used in "computation block 1.F". Now when we are testing predicate P_i then in all paths chosen for P_i variable X will be defined but not used. This leads to the following two cases:

Case one: X is an input variable.

In this case there will be a zero column in the path information matrix for input variable X. Since X is an input variable, it will not add anything to the blindness space for any of the predicates, therefore this variable will not affect the error space.

Case two: X is a program variable.

In this case there will be a row p and column q in the path information matrix corresponding to the program variable X.

Where,

$$a_{pq} = -1,$$
$$a_{kq} = 0,\ 1 \le k \le r \text{ and } k \ne p,$$
$$a_{pj} = 0,\ 1 \le j \le c \text{ and } j \ne q$$

r is the number of rows and c is the number of columns in the path matrix. Due to this there will be an assignment blindness vector given by expression "X". In order to eliminate this assignment blindness vector from the error space we have to find a path where the expression "X" does not evaluate to zero. It is obvious that no matter which path we choose for testing predicate P_i the expression "X" will always evaluate to zero. Therefore this vector cannot be eliminated from the error space. As a result this will increase the dimension of the final error space by one.

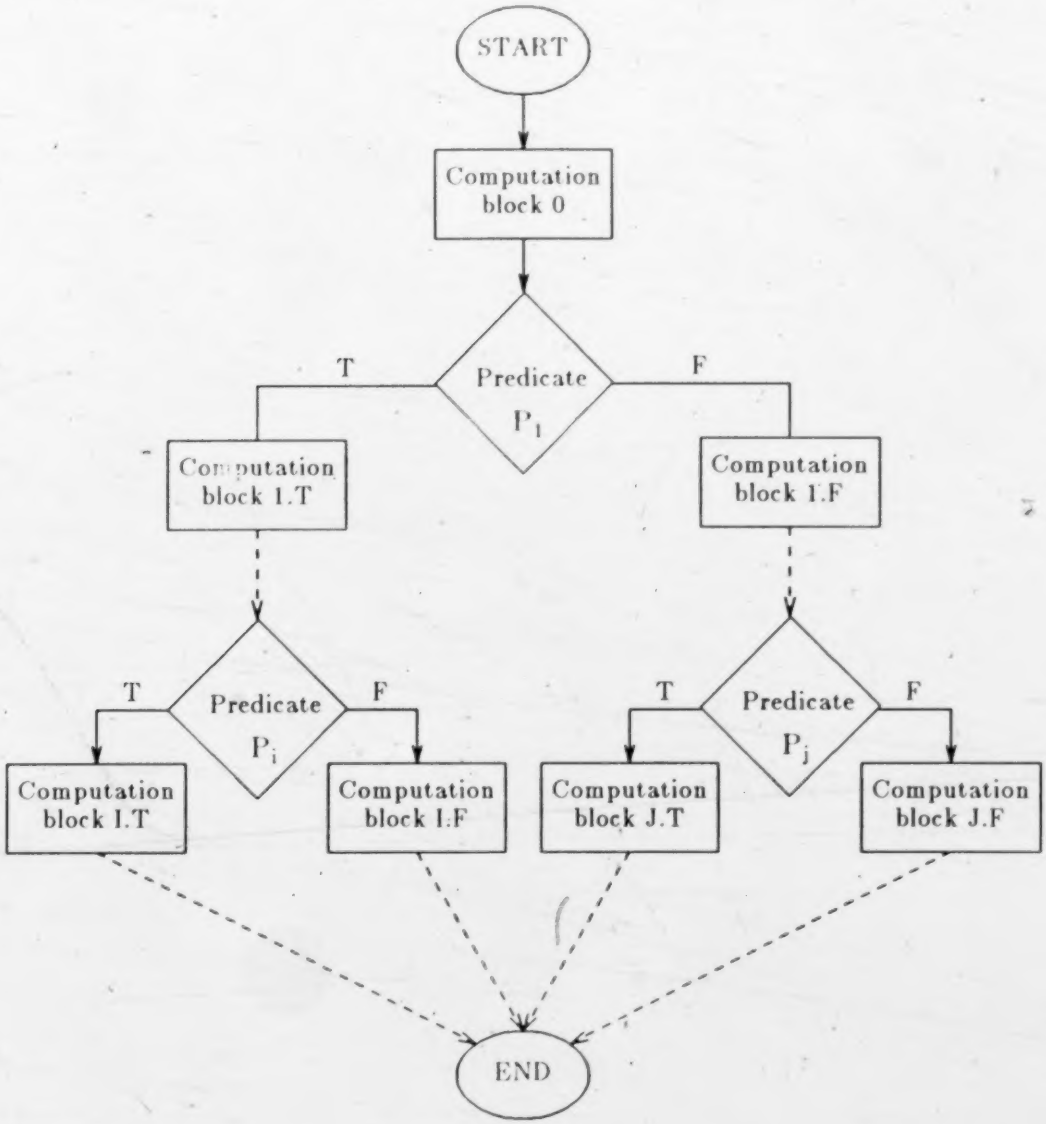

Figure 5.1 An Example Program Flowchart

The same explanation holds if the number of unused variables is more than one or if the program variable X is used and its value before the predicate P_i is zero.

5.2. Equality Predicates:

Every predicate encountered along a path places restrictions on the legal set of input values for that path. During the course of this research several experiments have been conducted in order to investigate the effects of equality restrictions. After performing several experiments it was found that in general if equality restrictions were encountered along a path then there is a possibility that some predicate error might go undetected. Clearly any linear combination of these predicate errors will also go undetected. In some cases these equality restrictions might increase the dimension of the error space and in some cases this will not affect the irreducible error space at all. The following example can be used to explain this.

This program given in Figure 5.2 has two input variables and one program variable. In this example the following are the possible control paths for testing predicate T3:

1. (T_1:f, T_2:t, T_3:?)
2. (T_1:f, T_2:f, T_3:?)
3. (T_1:t, T_2:t, T_3:?)
4. (T_1:t, T_2:f, T_3:?)

Here there is an equality restriction along paths 1 and 3 (i.e., predicate T_2 is true). In the case of path 1 it does not increase the dimension of the error space due to this equality restriction, because the expression for both the equality restriction

```
          READ, X, Y
          A = 0
(T1)      IF (Y .GT. X) THEN DO
            A = X
          ELSE DO
            A = Y
          END IF
(T2)      IF (A .EQ. 0) THEN DO
            PRINT, 'A is zero'
          END IF
(T3)      IF (Y .GT. 0) THEN DO
            PRINT, 'Y is positive'
          END IF
          STOP
          END
```

Figure 5.2 A Example Program (Program #3 in Table 4.1)

and target predicate are the same (i.e. "0 - Y"). On the other hand, in the case of path 3, the equality restriction increases the dimension of the irreducible error space by one.

In this example if we choose path 3 as the first path for testing predicate T3, then in order to achieve the irreducible error space (dimension is one), we have to test path 4 plus one more (either path 1 or 2). On the other hand if we choose any other path as our first path then we only need to test one more path in order to reduce the dimension of the error space to one. In Table 5.1 four different set of test paths which are sufficient to test predicate T3 are given. Here the initial dimension of the error space is 4.

In conclusion, if a path with equality restrictions is selected, then it might be possible that some predicate errors will go undetected along that path. Therefore, in such cases we would like to find a path without those equality restrictions. If such a path cannot be found then the dimension of irreducible error space is increased by one.

5.3. Invariant Expressions

The third category of vectors in the final error space is due to invariant expressions, and have constituted a bit of a surprise in the way they have been manifested in the experiments; these will be discussed more fully after the experimental results are described.

	Path	Undetected error space
Set # 1	3	3
	4	2
	1	1
Set # 2	4	2
	3	does not reduce
	1	1
Set # 3	1	2
	2	does not reduce
	3	does not reduce
	4	1
Set # 4	2	2
	1	does not reduce
	3	does not reduce
	4	1

Table 5.1 Summary of Sufficient Set of Paths for Testing Predicate T_3

5.4. Program Loops

Here we will consider programs of the following form:

```
<initialization statements>
LOOP <loop predicate>
   <loop body statements>
END LOOP
```

These programs tend to occur frequently in programming in order to accomplish some specific task, e.g., sort a table, traverse a data structure, calculate some arithmetic function, etc.

A program with loops is the most difficult type of program to test. The presence of a simple WHILE loop in a program may generate an infinite number of paths. Even when this is not the case, the number of paths for even simple programs can be prohibitively large. The program flowchart in Figure 5.3, for example, can easily be shown to contain over 2^{12} paths even though each loop can be executed no more than 12 times.

There are two types of loops which can occur in a program. First consider a loop which is iterated a fixed number of times. In other words, whenever we reach such a loop the very first time we know how many times we are going to iterate that loop. For example, a DO-loop

```
      <initialization statements>
      DO  stn i = m1, m2 [,m3]
         <loop body statements>
stn   CONTINUE
```

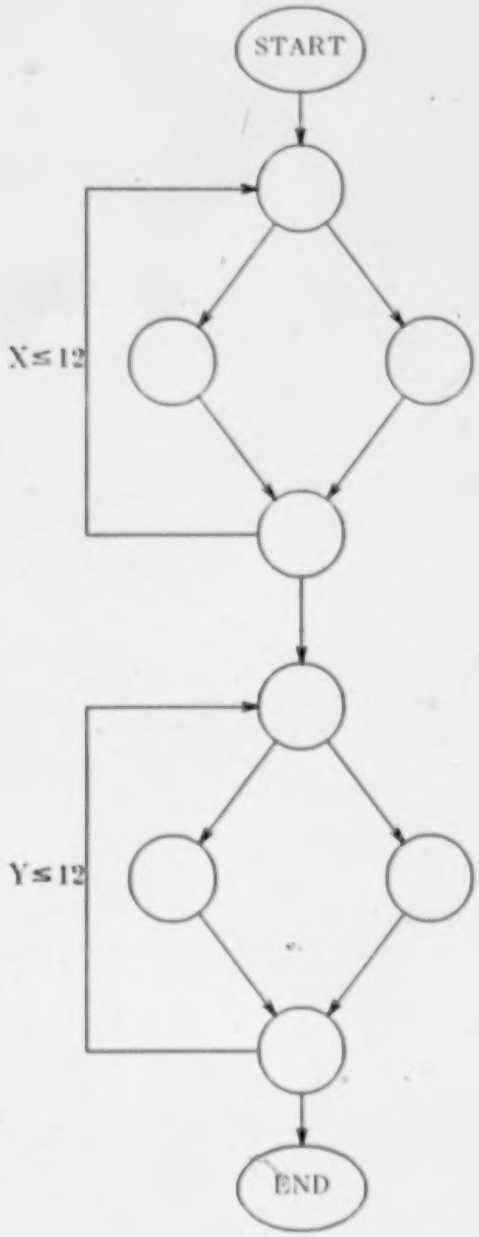

Figure 5.3 An Example Program Flowchart with 2^{12} Paths

stn

is the number of an executable statement appearing after the DO statement in the program unit containing the DO.

i

is an integer, real, or double precision variable (not an array element) called the DO variable.

m1, m2, and **m3**

are integer, real, or double precision arithmetic expressions. **m1** , **m2** and **m3** are intial value, test value and incremental value respectively. m3 is optional and cannot have a value of zero; if it is omitted, its value is assumed to be 1, and the preceding comma must be omitted.

The statements in the range of the DO are executed only if:

m1 is less than or equal to m2, and m3 is greater than 0

or,

m1 is greater than or equal to m2, and m3 is less than 0.

If one of the above relationships between m1, m2, and m3 is true, the first time the statements in the range of the DO are executed, i is initialized to the value of m1; on each succeeding iteration, i is modified by the value of m3. The number of iterations that can be executed, also called the iteration count, is the value of:

MAX (INT((m2 - m1 + m3) / m3), 0).

The first time i exceeds m2 at the end of the iteration, control passes to the statement following the statement numbered stn.

If at the first time one of the relationships is not true, then the statements in the range of the DO are executed and execution continues with the statement following the last statement of the range of the DO (i.e. the DO loop is executed

exactly one time).

The DO variable should not be redefined within the range of the DO-loop. However, any variables in the expressions for the initial value, test value, or incremental value may be redefined in the DO-loop without changing the iteration count as established for the DO statement.

The second type of loop is a WHILE loop, and is of the following form:

```
<initialization statements>
WHILE <loop predicate> DO

   <loop body statements>

END WHILE
```

The WHILE loop is a type of loop which can generate an infinite number of paths. The meaning of a WHILE-loop is: "while some condition is true, repeat this group of statements".

In order to investigate the iteration limit of a WHILE-loop a number of experiments with SPTEST were done. The program given in Figure 5.4 computes the greatest common factor of two integers. In this program, there are two input variables and three program variables. The following four paths are sufficient to test predicate T_1.

1. $(T_1{:}?)$
2. $(T_1{:}t, T_2{:}t, T_1{:}?)$
3. $(T_1{:}t, T_2{:}f, T_1{:}t, T_2{:}t, T_1{:}?)$
4. $(T_1{:}t, T_2{:}f, T_1{:}t, T_2{:}t, T_1{:}t, T_2{:}t, T_1{:}?)$

Here path one requires no iteration, path two one iteration, path three two

```
          PROGRAM GCF
          READ, A, B
          S = A
          T = B
          U = 0
(T1)      WHILE (S .NE. T) DO
(T2)        IF (S .GT. T) THEN
              S = S - T
            ELSE
              U = S
              S = T
              T = U
            END IF
          END WHILE
(T3)      IF (S .EQ. 1) THEN
            PRINT, A," and ",B," are relatively prime "
          ELSE
            PRINT, "The Greatest Common Factor of ",
                   A," and ",B," is ", S
          END IF
          END GCF
```

Figure 5.4 Euclid's Algorithm for GCF

iterations and path four three iterations. After testing these four paths the irreducible error space for predicate T_1 will be:

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ -1 \\ 1 \\ 0 \end{bmatrix}$$

Here the total undetected space is just the self−blindness vector. Since this error term can never be eliminated, this set of test paths forms a sufficient set of test paths for testing predicate T_1.

In order to test predicate T_2 a set of paths on which tests can be performed is selected by simply extending each test path from T_1 to T_2. The irreducible error is the same as for predicate T_1.

If we continue testing T_3 by simply extending each of the four test paths for T_1 (i.e. assuming the last interpretation of T_1 to be false so that T_3 is the next predicate to be chosen), the total space of undetectable errors for T_3 will be:

$$\begin{bmatrix} 0 & -1 \\ 0 & 0 \\ 0 & 0 \\ -1 & 1 \\ 1 & 0 \\ 0 & 0 \end{bmatrix}$$

Here the second vector is still the self blindness vector which cannot be eliminated. The second vector is the equality restriction (S = T). Since this relation must hold for any halting path for predicate T_3, the set of test paths for T_3 is sufficient. Table 5.2 summarizes the results of the experiment conducted with the

	Path	Undetected Error Space
For predicate T_1 and T_2	1	4
	2	3
	3	2
	4	1
For predicate T_3	1	5
	2	4
	3	3
	4	2

Table 5.2 Summary of Sufficient Set of Test Paths for the GCF Program

GCF program. Here the initial dimension of the error space is 6. In the GCF program if one of the input variables is a negative number and the other is a positive number, then the program computes an incorrect function. For example if,

```
A = -.01
B = .01
```

then the GCF of A and B should be .01. But when we test any of paths 2, 3, or 4, the answer is .02, which is incorrect. Therefore, the paths approved by SPTEST for testing a program is a good set of paths to detect this error.

The program shown in Figure 5.5 is a more complicated one. In this program there is one nested loop. Here there are two input variables and two program variables.

In this program the following paths are sufficient for testing predicate T_1:

1. $(T_1{:}?)$
2. $(T_1{:}t, T_2{:}t, T_2{:}f, T_3{:}f, T_1{:}?)$
3. $(T_1{:}t, T_2{:}f, T_3{:}t, T_3{:}f, T_1{:}?)$

In the first path there is no iteration of any of the loops, in the second path there is one iteration for each of predicates T_1 and T_2, and in the third path there is one iteration for each of predicates T_1 and T_3. After testing these three paths the total undetected space for predicate T_1 will be:

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ -1 \\ 1 \end{bmatrix}$$

There are no assignment statements between T_1 and T_2, and between T_1 and T_3. All of the predicates in this program have the same functional form, and also

```
          PROGRAM GCD
          READ, X, Y
          A = X
          B = Y
(T1)      WHILE (A .NE. B) DO
(T2)        WHILE (A .GT. B) DO
              A = A - B
            END WHILE
(T3)        WHILE (B .GT. A) DO
              B = B - A
            END WHILE
          END WHILE
          PRINT, X, Y, A, B
          END GCD
```

Figure 5.5 Euclid's Algorithm for GCD

there are no equalities. Therefore the same set of paths are also sufficient to test predicates T_2 and T_3. Table 5.3 summarizes the results of the experiment done with the GCD program. Here the initial dimension of the error space is 5.

Similar results are obtained from other experiments. Based on these results the following loop heuristic is proposed.

For any loop in the program, test only those paths which perform that loop no more than m+n iterations for testable subpaths.

5.5. Experimental Results

Table 5.4 shows the results of the experiments on the nine programs described in Table 4.1 in Chapter 4. For each program, the initial dimension of the error space is given together with the value of n, and the number of paths necessary to test all the predicates in the program. For each predicate T_i, the number of sufficient paths are given together with the reduction in the error space for each path; the dimension of the final irreducible error space is specified. When this dimension is larger than one, the vectors in this space are identified as unused variables, equality restrictions and invariant expressions.

Theorem 1 provides a bound of (n+e+1) for the sufficient paths; since the first paths are all chosen such that e=0, notice what a tight upper bound (n+1) provides for these programs, with this bound being achieved for some predicate in four of the nine programs. Note that in most cases, the number of paths required to test all predicates in the program is equal to that for testing the predicate requiring the largest number of paths; the only exceptions are programs #4 and #9, and even then only one more path is needed. Of course, in the experiments, paths were chosen so as not only to give a minimum number of sufficient subpaths for each

	Path	Undetected Error Space
	1	3
For predicates	2	2
T_1, T_2 and T_3	3	1

Table 5.3 Summary of Sufficient Set of Test Paths for the GCD Program

Program	Predicate	Number of Sufficient Paths	Intial Error Space	Reduction In Error Space	Irreducible Error Space	Vectors Other Than Self-Blindness	Paths Required for Entire Program
1	T_1	3	5	3,2,1	1		3
	T_2	3	(n=2)	3,2,1	1		
	T_3	3		3,2,1	1		
2	T_1	2	5	4,3	3	{Unused Var Inv Exp	2
	T_2	2	(n=3)	4,3	3	2 Inv. Exp.	
3	T_1	1	4	2	2	Unused Var.	2
	T_2	2	(n=1)	2,1	1		
	T_3	2		2,1	1		
4	T_1	1		3	3	2 Unused Var.	4
	T_2	1	5	3	3	2 Unused Var.	
	T_3	3	(n=2)	3,2,1	1		
	T_4	3		3,2,1	1		
5	T_1	4	6	4,3,2,1	1		4
	T_2	4	(n=3)	4,3,2,1	1		
	T_3	4		5,4,3,2	2	Equality	
6	T_1	1		6	6	5 Unused Var.	4
	T_2	2	8	6,5	5	{3 Unused Var Inv Exp	
	T_3	4	(n=5)	6,5,4,3	3	{Unused Var Inv Exp	
	T_4	4		6,5,4,3	3	{Unused Var Inv Exp	

Table 5.4 Results of Experiments Using SPTEST

Program	Predicate	Number of Sufficient Paths	Intial Error Space	Reduction In Error Space	Irreducible Error Space	Vectors Other Than Self-Blindness	Paths Required for Entire Program
7	T_1	4	14 (n=5)	6,5,3,1	1		5
	T_2	5		6,5,3,2,1	1		
	T_3	3		5,2,1	1		
	T_4	3		6,4,2	2	Equality	
	T_5	3		6,4,2	2	Equality	
8	T_1	1	17 (n=5)	6	6	5 Unused Var.	4
	T_2	1		6	6	{Unused Var 4 Inv. Exp	
	T_3	4		6,5,3,2	2	Inv. Exp.	
	T_4	3		6,4,3	3	{Unused Var Inv. Exp	
	T_5	3		6,4,3	3	{Unused Var Inv. Exp	
	T_6	3		6,4,3	3	{Unused Var Inv. Exp	
	T_7	4		6,5,4,3	3	{Unused Var Inv. Exp	
9	T_1	1	21 (n=8)	13	13	{9 Unused Var 3 Inv. Exp	6
	T_2	2		13,12	12	{8 Unused Var 3 Inv. Exp	
	T_3	5		12,10,8,6,2	2	Unused Var.	
	T_4	5		13,10,8,7,4	4	3 Unused Var.	
	T_5	5		13,10,8,7,4	4	3 Unused Var.	
	T_6	5		13,9,7,6,5	5	2 Unused Var.	
	T_7	5		13,9,7,6,5	5	2 Unused Var.	
	T_8	5		13,11,9,7,2	2	Equality	
	T_9	5		13,11,9,7,2	2	Equality	

Table 5.4 cont.

predicate, but also so that these subpaths could be extended to paths which test all predicates in the program.

These experiments have shown that it is not unusual for predicate irreducible error spaces to have dimension greater than one; the most common reason for these additional vectors is due to unused variables. Yet there a number of occurrences of vectors corresponding to both equality restrictions and invariant expressions.

From program #6 (given in Figure 3.2), notice from Table 5.4 that there is an invariant expression in the irreducible error space for predicates T_2, T_3 and T_4. In Figure 5.6, here the irreducible error space for T_3 is of dimension three. The second column vector represents an unused variable F and the third column vector represents the self-blindness vector for predicate T_3, I = Y; the first column vector represents the invariant expression

$$D = 2*X + Y, \tag{5}$$

which occurs as line 12 of program #6. However, if we examine the irreducible error space for predicate T_4 also shown in Figure 5.6, we might get confused. We can identify the second column vector as D = 2, the self-blindness vector for T_4, and the third column again as the unused variable F. The first column vector corresponds to an expression

$$2*X + Y = 2, \tag{6}$$

which seems confusing. However, recall that this is a vector space, and if we subtract the vector (D − 2) from equation (6), we will generate the invariant expression (5), the same as in T_3. This illustrates the challenge in detecting and identifying invariant expressions or equality restrictions.

Figure 5.7 shows another interesting case of an invariant expression, in this case a loop invariant. The irreducible error space for predicate T_1 contains

Irreducible error space after testing predicate T_3.

Const	0	0	0
X	-2	0	0
Y	-1	0	-1
C	0	0	0
D	1	0	0
E	0	0	0
F	0	1	0
I	0	0	1

Invariant expression D = 2*X + Y

Irreducible error space after testing predicate T_4.

Const	-2	-2	0
X	2	0	0
Y	1	0	0
C	0	0	0
D	0	1	0
E	0	0	0
F	0	0	1
I	0	0	0

Figure 5.8 Example of an Invariant Expression for Pragram #6

```
                READ, N
                I = 0
                J = N
                R = 0
(T1)            WHILE (J .GE. 1.0) DO
                  I = I + 1
                  J = N - I
                END WHILE
                R = N - 1
(T2)            IF (R .GE. 0.50) THEN DO
                  I = I + 1
                END IF
                PRINT, N, I
                END
```

Program 2: Integer Round - Up.

Irreducible error space after testing two paths for predicate T_1 is:

Const	1	-1	0
N	-1	0	0
I	1	0	0
J	0	1	0
R	0	0	1

Figure 5.7 Example of a Loop Invariant for Program #2

self-blindness vector J = 1 as the second column vector, and the third column vector is the unused variable R. The first column vector corresponds to the expression (I = N − 1), which is a loop invariant corresponding to T_1, and does not occur explicitly as any expression within the program.

These observations and experience have allowed us to formulate some guidelines as to how a sufficient set of paths to test predicates should be chosen. It is important that a static analysis should be done so as to assure that all variables are defined before being used, and after being defined, are actually used. If either condition is violated, certain contradictions will show up in the predicate error space analysis.

If a small number of sufficient paths are desired for a given predicate, then the first path should be carefully selected. Theorem 1 has indicated that one guideline is to choose the first path with the fewest possible equality restrictions. In selecting subsequent paths for that predicate, the vectors in the remaining error space should be carefully analyzed, and paths selected which tend to eliminate those vectors; for example, in the case of unused variables in this error space, try to select a path which will use those variables. With other vector expressions or equality restrictions, try to select a path where those expressions or restrictions do not hold.

As for the selection of predicates to test, predicates with the fewest paths leading to them should be chosen first; these tend to be predicates near the beginning of the program. In order to obtain a small overall number of paths to test all the predicates, the paths should be considered which are extensions of subpaths previously selected for earlier predicates. This should still be done so as to eliminate as many vectors in the error space as possible. The experiments in Table 5.4 showed that this approach was quite successful. It can be observed that the same vectors tend to appear from one predicate error space to another, and illustrate why this

strategy is reasonable and successful.

Iteration loops pose special problems, because there may be an infinite number of paths to consider; a stopping condition for path testing is needed. Here is where analysis of the remaining vectors in the predicate error space can either indicate an appropriate path to choose or a stopping condition. If no path can be found to eliminate vectors in the error space, the example from Figure 5.7 illustrates that we should look for invariants which will terminate this search for additional paths.

In summary, we can give either partial or complete answers to the questions posed in Chapter 4:

(1) It is not possible to find a minimum set of paths for testing a given predicate because of the potentially infinite number of paths to examine with iteration loops and also because the problem is essentially NP-hard.

(2) Both analytical and experimental results have shown that the number of paths required to sufficiently test a predicate is far less than the bound of (m+n); a good bound has been shown to be (n+e+1), where e is the number of equality restrictions associated with the first path selected for that predicate.

(3) It would appear that it is possible to reduce the number of paths for testing a predicate by carefully choosing the first path. This path should be associated with the fewest equality restrictions. Subsequent paths should be chosen so as to eliminate as many vectors as possible in the remaining error space.

(4)-(5) It appears that we can substantially reduce the number of paths to test all the predicates far below the upper bound of p(m+n). As a matter of

fact, by utilizing common subpaths for testing subsequent predicates, and by choosing predicates which occur early in the program to evaluate first, our experiments have shown that the total number of paths for testing all the predicates is either equal to the maximum of (n+e+1) over all predicates, or only sightly larger. It remains for further experimentation or analysis to show if this will hold in general.

(6) We have found experimentally that it is not unusual for an irreducible error space to have dimension well beyond one, and this ranges up to 13 for predicate T_1 in program #9. We have been able to characterize vectors other than the self-blindness vectors as those corresponding to unused variables, equality restrictions, and invariant expressions. To a great extent we should be able to predict a priori the first two types of vectors (except possibly for coincidental equalities), but invariant expressions will require further study.

Chapter 6

Conclusions

6.1. Overview

This research makes the following contributions to knowledge on program testing:

1) a computer system for selecting a sufficient set of test paths for testing a computer program predicate;

2) a set of experiments to show that it is possible to select a set of test paths for testing a restricted but powerful class of programs;

3) heuristics for selecting best test paths for testing computer program.

6.2. Summary

Over the years there have been many approaches proposed for program testing. Many of these proposed testing methods fall within a class of strategies called **path analysis testing,** where the actual testing process is conducted in the following two steps:

1. selection of a path or set of paths along which testing is to be conducted, and

2. selection of input data to serve as test cases which will cause the chosen paths to be executed.

This research is concerned with the first step of path analysis testing, i.e., the selection of test paths. In reference [22,23] Zeil and White have developed a model

for predicate testing. In this model they have demonstrated that the class of possible errors forms a vector space; by using this vector space measure it is possible to characterize undetected predicate errors. In this research we have used this model to investigate the following issues:

1) After a number of paths have been tested, what is the marginal advantage of choosing yet another path?

2) Is there a point at which we may say that no more paths need be chosen through some program construct to be tested, i.e., that is has been sufficiently tested ?

A computer system based on this model is implemented on UNIX and MTS. This system can be used to select test paths for testing a Fortran program. Once a set of paths are selected then any reliable testing strategy can be used to conduct the actual testing along these selected test paths.

In order to better understand the path selection process, a series of experiments using this system were conducted. Based on the results obtained from these experiments the following heuristics for selecting test paths are proposed:

1. For testing any predicate, carefully choose the first path such that the first path has fewest equality restrictions.

2. By utilizing common subpaths for testing subsequent predicates and by choosing predicates which occur early in the program to evaluate first, the total number of paths required for testing all the predicates in a program can be substantially reduced.

3. If possible after selecting the first path, select those paths as subsequent paths which employ unused program variables.

4. For any loop in the program, test only those paths which execute that loop no more than m+n iterations.
5. The next path can be chosen by carefully examining the spanning vectors of the error space for the previously selected paths.
6. By carefully examining the spanning vectors of the error space for all selected paths, one can determine whether or not there is a need to select yet another path.

These experiments have also shown that it is not unusual for an irreducible error space to have dimension well beyond one.

In summary, this research has made an attempt to better understand the path selection process. One of the prime objectives of this research has been to design and implement a path selection system. However, many of the related issues need further research. Some of these issues are described in the next section.

6.3. Further Research

It is clear that further experimentation is needed; it has been interesting to us that small programs, such as programs #2 and #6, can give considerable insight rather than very large programs. We need to better understand loop invariant expressions and why they appear in some irreducible error spaces and not in others. It would be desirable to extend the experiments to programs with compound predicates.

In further experimentation the system needs to be extended in several ways. One considerable extension would be from linearly domained programs to vector-bounded programs; this will pose extra problems for determining path feasibility for

this class of programs, because path feasibility could always be determined for linearly domained programs. Another objective would be to modify the system so as to be able to accommodate Zeil's extensions for computation errors and assignment errors (which complement predicate errors as the other type of domain error). Zeil's techniques for perturbation testing [11-12] can be explored for this extension, together with the error spaces and paths required to test these constraints.

At present this system is used in a research context, and as such is impractical in its use of CPU time, requiring seconds of supermini (VAX 11/780) computation time. This is unacceptable in an operational setting, and path generation will have to be generated much more efficiently.

SPTEST is now only available as an aid to the user in selecting paths. Extensions as those suggested above can then lead to an automated system for selecting paths for path - oriented testing techniques, including domain testing. It is possible that some of these testing ideas can be applied to other testing approaches as well.

References

1. B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold Company, 1983.

2. L. A. Clarke, "Automatic Test Data Selection Techniques", *Infotech State of the Art Report on Software Testing*, Vol. 2., 1979, .

3. E. A. Cohen, *A Finite Domain-Testing Strategy for Computer Program Testing*, Ph.D. Dissertation, The Ohio State University, Columbus, Ohio, 1978.

4. J. L. Elshoff, "An Analysis of Some Commercial PL/I Programs", *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 2, June 1976, pp. 113-120.

5. L. D. Fosdick and L. J. Osterweil, "Data Flow Analysis in Software Reliability", *ACM Computing Surveys*, Vol. 8, No. 3, September 1976, pp. 227-231.

6. J. B. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection", *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975, pp. 156-173.

7. W. E. Howden, "Methodology for the Generation of Program Test Data", *IEEE Transactions on Computers*, Vol. C-24, No. 5, May 1975, pp. 554-559.

8. W. E. Howden, "Reliability of the Path Analysis Testing Strategy", *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 3, September 1976, pp. 208-215.

9. W. E. Howden, Introduction to the Theory of Testing, in *Tutorial: Software & Valodation Techniques*, E. F. Miller, Jr. and W. E. Howden (ed.), IEEE Computer Society, 1978, pp. 16-19.

10. E. F. Miller, Jr., Software Testing Technology: An Overview, in *Handbook of Software Engineering*, C. R. Vick and C. V. Ramamoorthy (ed.), Van Nostrand Reinhold Company, 1984.

11. D. E. Knuth, "An Empirical Study of FORTRAN Programs", *Software - Practice and Experience*, Vol. 1, No. 2, April-June 1971, pp. 105-133.

12. Z. Manna and R. Waldinger, "The Logic of Computer Programming", *IEEE Transactions on Software Engineering*, Vol. SE-4, 1978, pp. 199-229.

13. T. J. McCabe, "A Complexity Measure", *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, December 1976, pp. 308-320.

14. C. V. Ramamoorthy, S. F. Ho and W. T. Chen, "On the Automated Generation of Program Test Data", *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, December 1976, pp. 293-300.

15. S. H. Saib, Application of the Software Quality Laboratory, *State of the Art Report on Program Testing*, 1979.

16. L. J. White, F. C. Teng and D. W. Coleman, An Error Analysis of the Domain Testing Strategy, Tech. Rep.-78-2, Computer and Information Science Research Center, The Ohio State University, Columbus, Ohio, August 1978.

17. L. J. White, E. I. Cohen and B. Chandrasekaran, A Domain Strategy for Computer Program Testing, Tech. Rep.-78-4, Computer and Information Science Research Center, The Ohio State University, Columbus, Ohio, August 1978.

18. L. J. White and E. I. Cohen, "A Domain Strategy for Computer Program Testing", *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 3, May 1980, pp. 247-257.

19. L. J. White, Basic Mathematical Definitions and Results in Testing, in *Computer Program Testing*, B. Chandrasekaran and S. Radicchi (ed.), North-Holland Publishing Company, 1981.

20. L. J. White and P. N. Sahay, Experiments Determining Best Paths for Testing Computer Program Predicates, *Proceedings of the Eighth International Conference on Software Engineering*, London, UK, August 1985, pp. 238-243.

21. L. J. White and P. N. Sahay, A Computer System for Generating Test Data Using The Domain Strategy, Tech. Rep. 85-6, Department of Computing Science, The University of Alberta, February 1985.

22. S. J. Zeil and L. J. White, Selecting Sufficient Sets of Test Paths For Program Testing, *Proceeding of the Fifth International Conference on Software Engineering*, 1981, pp. 184-191.

23. S. J. Zeil, Selecting Sufficient Sets of Test Paths For Program Testing, OSU-CISRC-Tech. Rep.-81-10, The Ohio State University, Columbus, Ohio, 1981.

24. S. J. Zeil, "Testing for Perturbations of Program Statements", *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 3, May 1983, pp. 335-346.

25. S. J. Zeil, Perturbation Testing for Computation Errors, *Proceedings of the Seventh International Conference on Software Engineering*, March 1984.

Appendix 1

Current Implementation of SPTEST

Implemented	Not Implemented
ACCEPT	ASSIGN
ASSIGNMENTS STMTS	BACKSPACE
AT END DO	BLOCK DATA
CASE	CALL
COMPUTED GO TO	CHARACTER
CONTINUE	COMMON
DO CASE	COMPLEX
DO	DATA
END	DIMENSION
END AT END	DUMPLIST
END BLOCK	DOUBLE PRECISION
END CASE	ENDFILE
END IF	ENTRY
END WHILE	EQUIVALENCE
ELSE DO	EXTERNAL
EXECUTE	FUNCTION
FORMAT	NAMELIST
GO TO	ON ERROR GO TO
IF(...)<EXE STMT>	PAUSE
IF NONE DO	PUNCH
IF(...)THEN DO	RETURN
PRINT	REWIND
READ(FORMATTED)	SENSELIGHT
READ(UNFORMATTED)	TYPE
REMOTE BLOCK	IMPLICIT
STOP	INTEGER
WHILE(...)DO	LOGICAL
WRITE	REAL

Maximum number of lines in a program is 150.

Restrictions on constructions:

Type	Maximum Number
Arithmetic Statements	100
Assignment Statements	50
Computed Go To	3
Do Loops	25
Input Variables	20
Output Variables	20
Labels	40
Predicates	100
Read Statements	20
Write Statements	20
Remote Blocks	10

Appendix 2

Using the Sufficient Path Testing System

Both the Domain Testing System and Sufficient Path Testing System are located in UNIX on Cavell under directory **/u1/grad/sahay/program.testing** at the University of Alberta. The source code for the first phase of SPTEST is located under the directory **domain/src** and for the second and third phase the source code is located under **sptest/src.**

In order to use the Sufficient Path Testing System, the user has to go to the directory **/u1/grad/sahay/program.testing/sptest.** There are three command files which have been set up to allow the user a variety of options in using the system. By typing "takeab <input file>" both parts of the system will be executed. At first the system prints "Compiling <input file>", and as each statement of the program compiles, the system prints "Line XX Translated". After the program is compiled the system prints "<input file> Compiled".

After that second phase is invoked and the system prints "Reading Tables". After the program tables have been read the system prints "Ready" and user is asked to select a predicate for testing. After that it starts line-by-line processing of the input program. The user now has the chance of choosing a subpath for testing the target predicate. Once a subpath is selected then SPTEST is called. The user is asked to indicate whether or not this path is the first path for that predicate. The system evaluates the usefulness of that path and the user is notified. At this point the user is either asked to select another path for that predicate or notified that a sufficient set of paths for testing that predicate has been obtained.

List of options:

takeab <input file>

takea <input file>

takeb

The information generated by the system about the input program are located under directory **/u1/grad/sahay/program.testing/File** and are listed below:

Type	File Name	Generated By
List of Program	**program.lst**	First Phase
List of Predicates	**program.prd**	Second Phase
List of Decisions	**program.dec**	Second Phase
Spanning Vectors	**span.vector**	Second Phase

If a program is new and has not been tested on the system then the user should check the list of restrictions on FORTRAN programs (Appendix I). To make sure that the program has compiled correctly the user should use command "takea" and compare "program.lst" to the input program.

Appendix 3

Input Programs

The input programs used for the experimentation in Chapters 4 and 5 are listed in this appendix.

Program 1: Euclid GCD

```
READ, X,Y
A = X
B = Y
WHILE (A .NE. B) DO
  WHILE ( A .GT. B) DO
    A = A - B
  END WHILE
  WHILE ( B .GT. A) DO
    B = B - A
  END WHILE
END WHILE
PRINT, X,Y,A
STOP
END
```

Program 2: Integer Round-up

```
READ, N
I = 0
J = N
R = 0
WHILE (J .GE. 1.0) DO
  I = I + 1
  J = N - I
END WHILE
R = N - I
IF (R .GE. 0.50) THEN DO
  I = I + 1
END IF
PRINT, N, I
STOP
END
```

Program 3:

```
READ, X, Y
A = 0
IF (Y .GT. X) THEN DO
  A = X
ELSE DO
  A = Y
END IF
IF (A .EQ. 0) THEN DO
  PRINT, A
END IF
IF (Y .GT. 0) THEN DO
  PRINT, Y
END IF
STOP
END
```

Program 4: Integer Division Remainder

```
READ, X, Y
R = 0
A = 0
IF (X .GE. 0) THEN DO
  IF (Y .GT. 0) THEN DO
    R = X
    WHILE (R .GE. Y) DO
      A = Y
      WHILE (R .GE. A) DO
        R = R - A
        A = A + A
      END WHILE
    END WHILE
  END IF
END IF
PRINT, R, X, Y
STOP
END
```

Program 5: Euclid GCF

```
ACCEPT,A,B
S=A
T=B
U=0
WHILE (S .NE. T) DO
  IF (S .GT. T) THEN DO
    S=S-T
  ELSE DO
    U=S
    S=T
    T=U
  END IF
END WHILE
IF (S.EQ.1) THEN DO
  PRINT, A,B
ELSE DO
  PRINT, A ,B,S
END IF
STOP
END
```

Program 6: Conditional Series Summation

```
READ, A, B
C = 0
D = 0
E = 0
F = 0
I = 0
IF (A .GT. B) THEN DO
  C = B + 1
ELSE DO
  C = B - 1
END IF
D = 2 * A + B
IF (C .GT. 0) THEN DO
  I = 1
  WHILE (I .LE. B) DO
    E = E + 2 * I
    I = I + 1
  END WHILE
END IF
IF (D .LE. 2) THEN DO
  F = E + A
ELSE DO
  F = E - A
END IF
PRINT, F
STOP
END
```

Program 7: Sorted Set Intersection

```
      READ, M,N
      DO 10 I = 1, M
        READ, S1(I)
10    CONTINUE
      DO 20 I = 1, N
        READ, S2(I)
20    CONTINUE
      DONE = 0
      I = 1
      J = 0
      X = 0
      Y = 0
      WHILE (I .LE. M) DO
        IF (N .GT. 0) THEN DO
          J = 1
          DONE = 0
          WHILE (DONE .EQ. 0) DO
            X = S1(I)
            Y = S2(J)
            IF (X .EQ. Y) THEN DO
              PRINT, S1(I)
              DONE = 1
            END IF
            J = J + 1
            IF (J .EQ. N) THEN DO
              DONE = 1
            END IF
          END WHILE
        ELSE DO
          I = M + 1
        END IF
        I = I + 1
      END WHILE
      STOP
      END
```

Program 8: Binary Search

```
      READ, N
      DO 10 I = 1, N
        READ, B(I)
10    CONTINUE
      READ, A
      I = 0
      HIGH = 0
      LOW = 0
      MID = 0
      TEMP = 0
      IF (N .GT. 0) THEN DO
        HIGH = N + 1
        LOW = 1
        MID = (HIGH + LOW) / 2
        I = 0
        TEMP = B(MID)
        IF (A .EQ. TEMP)  THEN DO
          I = 1
        END IF
        WHILE (I .NE. 1)  DO
          TEMP = B(MID)
          IF ( A .EQ. TEMP) THEN DO
            I = 1
          ELSE DO
            IF ( A .LT. TEMP) THEN DO
              HIGH = MID
            ELSE DO
              LOW = MID
            END IF
            MID = (HIGH + LOW) / 2
          END IF
          IF (LOW .EQ. MID) THEN DO
            I = 1
          END IF
        END WHILE
        TEMP = B(MID)
        IF (A .EQ. TEMP) THEN DO
          PRINT, A
        END IF
      END IF
      STOP
      END
```

Program 9: Sorted Set Union

```
      READ, M, N
      DO 10 I = 1, M
        READ, S1(I)
10    CONTINUE
      DO 20 I = 1, N
        READ, S2(I)
20    CONTINUE
      J = M + N
      DO 30 I = 1, J
        A(I) = 0
30    CONTINUE
      I = 1
      J = 1
      K = 1
      X = 0
      Y = 0
      DONE = 0
      IF (N .LE. 0) THEN DO
        DONE = 1
      END IF
      IF (M .LE. 0) THEN DO
        DONE = 1
      END IF
      WHILE (DONE .NE. 1) DO
        X = S1(I)
        Y = S2(J)
        IF (X .EQ. Y) THEN DO
          A(K) = X
          I = I + 1
          J = J + 1
        ELSE DO
          X = S1(I)
          Y = S2(J)
          IF (X .LT. Y) THEN DO
            A(K) = X
            I = I + 1
          ELSE DO
            A(K) = Y
            J = J + 1
          END IF
        END IF
        K = K + 1
        IF (I .GT. N) THEN DO
          DONE = 1
        END IF
        IF (J .GT. M) THEN DO
          DONE = 1
        END IF
      END WHILE
```

```
      WHILE (J .LE. M) DO
        A(K) = S2(J)
        K = K + 1
        J = J + 1
      END WHILE
      WHILE (I .LE. N) DO
        A(K) = S1(I)
        K = K + 1
        I = I + 1
      END WHILE
      K = K - 1
      DO 40 I = 1, K
        PRINT, A(I)
   40 CONTINUE
      STOP
      END
```